MACROMODELING AND CHARACTERIZATION OF FILESYSTEM ENERGY CONSUMPTION FOR DISKLESS EMBEDDED SYSTEMS

A Thesis

by

SIDDHARTH CHOUDHURI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2003

Major Subject: Computer Engineering

MACROMODELING AND CHARACTERIZATION OF FILESYSTEM ENERGY CONSUMPTION FOR DISKLESS EMBEDDED SYSTEMS

A Thesis

by

SIDDHARTH CHOUDHURI

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Rabi Mahapatra (Chair of Committee)	Lawrence Rauchwerger (Member)
A. L. Narasimha Reddy (Member)	Valerie E. Taylor (Head of Department)

August 2003

Major Subject: Computer Engineering

ABSTRACT

Macromodeling and Characterization of Filesystem Energy Consumption for Diskless Embedded Systems. (August 2003)

Siddharth Choudhuri, Bachelor of Engineering, Sambalpur University

Chair of Advisory Committee: Dr. Rabi Mahapatra

The use and application of embedded systems in everyday life has proliferated in the past few years. These systems are constrained in terms of power consumption, available memory and processing requirements. Typical embedded systems like handheld devices, cell phones, single board computer based systems are diskless and use flash for secondary storage. The choice of filesystem for these diskless systems can greatly impact the performance and the energy consumption of the system as well as lifetime of flash.

In this thesis work, the energy consumption of flash based filesystems has been characterized. Both the processor and flash energy consumption are characterized as a function of filesystem specific operations. The work is aimed at helping a system designer compare and contrast different filesystems based on energy consumption as a metric. The macromodel can be used to characterize and estimate the energy consumption of applications due to filesystem running on flash.

The study is done on a StrongARM based processor running Linux. Two of the popular filesystems JFFS2 and ext3 are profiled.

To My Parents

ACKNOWLEDGMENTS

I would like to thank Dr. Rabi Mahapatra for all the help and guidance that he offered. I am grateful to him for letting me take up this interesting work.

Di Wu and Dr. Mahapatra's help in coming up with the experimental setup and LART board saved me a lot of time. I would like to thank Praveen and Junyi for their technical help, numerous advise and proof-reading.

I have had a memorable time working in the embedded systems and codesign group. I wish to thank John, Brenna, Mily, Anand, Abhijit and Ian for making grad life memorable with discussions ranging from life to quantum computing.

This work would not have been possible without help from the linux-mtd , arm-linux and kernel-newbies mailing lists. I would like to thank Dr. Reddy and Dr. Rauchwerger for serving on my committee. Their graduate classes have been an excellent learning experience for me.

Finally, I would like to thank people who are closest to me: my parents, who have been an infinite source of encouragement for me; and my sister and brother-in-law for their encouragement.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

The use and application of embedded systems in everyday life has proliferated in the past few years. The decreasing cost of processor based systems coupled with Moore's law has led to the development of a plethora of embedded devices. Embedded systems are ubiquitous and have become integral part of our daily lives. Some of the embedded systems that we come across in our day to day lives are in automated gas stations, airport kiosks, cash registers in grocery stores, microwaves, cell phones, ATM machines and PDAs to name a few. However, these systems, unlike conventional desktop systems are constrained in terms of power consumption, available memory, processing requirements and operating environments. These restrictions are mainly due to the fact that embedded systems are designed to serve specific purpose in mind and usually have a strict restriction on their form factor (physical size), computational capacity and cost of manufacturing. For example, a handheld should be designed to consume minimum power in order to have a prolonged battery life. Also, it should be as small as possible in its form-factor.

The *de-facto* standard for secondary storage in conventional desktop systems has been hard disk drive. However, constraints due to form-factor and power consumption rules out the option of using hard disk drive for secondary storage in embedded systems. Hard disk drives consume power in the order of hundreds of milli Watts [2] which is significant considering that most embedded systems run on battery. This has lead to the widespread use of a ROM, EEPROM or Flash based devices as means of secondary storage. The limitation of ROM and EEPROM is that they can be used as readonly systems whereas

---

The journal model is *IEEE Transactions on Automatic Control.*

flash based storage can be used as read-write systems. While read-only storage is suited for applications like mp3 players, most of the embedded systems like hand held devices have the need to be able to read and write data out of a secondary storage device, hence flash based storage is ideally suited. Flash memory is a type of constantly-powered nonvolatile memory that can be erase and reprogrammed in units of memory called blocks. Typical embedded systems like PDAs, cell phones and single board comuter based systems are diskless and use flash for secondary storage [3].

In the past few years, design efforts to minimize energy consumption of embedded systems have been of paramount importance. One of the reasons is due to the fact that the improvement in battery technology has lagged behind that of processor and digital systems [4]. However, the importance in power aware design is due to obvious reasons of making devices that consume less power leading to longer battery life. Conventional techniques of reducing power consumption have been studied and implemented in hardware [5]. The power dissipation in a CMOS circuit is due to[6]

1. Static dissipation due to leakage current. The total static power dissipation of a CMOS circuit is given by

$$P_s = \sum_{1}^{n} leakage\ current * supply\ voltage \tag{1.1}$$

   where, n = number of devices

2. Dynamic dissipation due to switching and capacitance which is given by

$$P_d = C_l * V_{DD}^2 * f_p \tag{1.2}$$

   where, C_l is the swithced capacitance, V_{DD} is the supply voltage and f_p is the system clock frequency.

The underlying ideas behind power aware designs have been to either reduce the operating

voltage or the frequency in (1.2) to reduce overall power consumption.

More recent techniques have revolved around exploiting software techniques to design energy efficient systems [7]. The obvious advantage offered by software techniques is the flexibility it offers. Software techniques do not require a re-design of hardware which is one of its biggest strengths as opposed to low level hardware techniques. Also, software techniques can be changed or dynamically tuned depending on application.

A. Operating Systems and Energy Consumption

In recent years, considerable work has been done in the area of Operating System directed power management for embedded systems. These studies have revealed various aspects of Operating Systems that can help in designing energy efficient embedded systems [8]. The various aspects of Operating Systems like scheduling, memory management, dynamic power management and idling of devices for power saving has been extensively studied in [9, 10, 11, 12, 13]. However, energy consumption from a filesystem perspective has not been done by any of the above. Another important reason why Operating System directed energy management has gained considerable importance is due to the fact that it is in the right place between high level applications and low level architecture. It is faster (in terms of time to market) and easier to make changes to software and reinstall, compared to change in hardware architecture which leads to entire redesign.

B. Macromodeling Filesystem Energy Consumption

Macromodeling is a technique that gives a high level model or equation for the system under study. In the case of filesystem energy consumption, the goal is to come up with mathematical formulae that describe the energy consumption of CPU and Flash due to filesystem related operations. (For example, creating a new file, removing directory, reads

and writes to a file etc). These equations would relate energy consumption as a function of filesystem metrics. The macromodel is derived after running experiments on an actual setup and then using standard techniques like regression analysis to generate a model.

C. Motivation

One of the important design decisions for an embedded systems developer is the energy consumption of the intended device. The energy consumption is an indicator of how fast the device discharges battery and obviously is an important design factor. However, fast time to market does not give the designer sufficient time to compare and contrast the energy consumption and performance of different filesystems. Also, a study of filesystem energy consumption would require an experimental setup, which would add to the cost and time to market. The aim of this paper is to give a quantitative study filesystems for a flash based device and provide with a study of energy consumption and performance based on standard benchmarks. We then provide a macromodel that can be used to relate filesystem usage with energy consumption. The motivation behind this macromodel is that an embedded systems developer can use this macromodel to estimate the energy consumption of filesystems at design time. The macromodel would have characteristics of energy consumed by both processor as well as flash. The macromodel could be used to make a tradeoff between available filesystem choices for energy aware systems.

To our knowledge, this is the first time filesystem energy characterization has been done for flash based devices. This can be used efficiently by a system designer to make filesystem tradeoffs and to save time without the need for an actual power measurement setup.

D. Contributions of the Thesis

1. Characterization of energy consumption overhead in an embedded system due to filesystem. In our work, we have isolated the filesystem energy consumption for a diskless embedded system from other components of the operating system. This study gives us a picture of overhead the filesystem imposes on the overall energy consumption.

2. A macro-model that describes filesystem energy consumption in terms of CPU and Flash. This model is a set of mathematical equations derived from macromodeling using regression analysis.

CHAPTER II

BACKGROUND AND PROPOSED RESEARCH

This chapter summarizes the current state of research work being done in the area of energy characterization for embedded operating systems and related work. It also introduces our methodology and discusses in detail the experimental setup required for the proposed research work.

A. Introduction

As discussed in the previous chapter, in the past few years research has focussed on studying, analyzing and finding ways to make operating systems energy efficient and power aware for embedded systems. Various subsystems of operating systems that can be improved towards a power aware system have been studied and introduced in [8]. Most of the approaches towards making operating systems power efficient have revolved around making changes to the scheduler and idling devices when they are not in use. A first step towards energy characterization of embedded systems was done in [9]. In this paper, the authors gave a per-instruction level energy consumption that could be used by software to characterize its power consumption. Characterizing energy consumption is important for two reasons

1. It gives an analytical model that can be studied and interpreted. Also, trends of energy consumption can be concluded from an anlalytic model.

2. It helps in developing tools that can be used to simulate and predict energy consumption of applications to a fair amount of accuracy, without having an actual measurement setup.

3. Having a macromodel is faster than conventional instruction level profiling or hard-

ware simulations.

1. Macromodeling

Macromodeling of operating systems energy consumption has been done by [14]. In this paper, the authors have used linear regression analysis to relate the operating system activities and system calls to energy consumption. For example, context switch energy consumption has been found out to be 12570 nJ for an ARM processor. However, this work does not take into consideration the type of filesystem present on the system and the measurements and macromodel developed is only for CPU energy consumption. In practical embedded systems, power is consumed both by the CPU and other devices interacting with the processor. A more general approach towards high level macromodeling has been described in [15]

2. Tools

Jouletrack is a web based tool developed by [16]. This tool gives the energy consumption and some other profiled data for a C program running on an ARM processor. The energy consumption is calculated from an instruction level energy consumption data available for ARM based processor. The tools is for profiling the energy consumption of statically linked C programs only. It does not give energy consumption from an operating system point of view. EMSIM is another energy profiling tool developed by [17]. This tool simulates a system from bootup having the required application running on as a ramdisk image. This tools gives a detailed results of energy consumption at system call level. However, only one application can be run on this tool. Another tool, called wattch has been developed based on simplescalar that characterizes the power consumption of an ARM processor [18, 19]

B. Problem Specification

The goal of the thesis is to characterize the energy consumption of the embedded system as a function of CPU and flash energy consumption. In other words, we want to obtain a mathematical model of the form

$$E_{system} = f(E_{cpu}, E_{flash}) \tag{2.1}$$

$$E_{cpu} = g(a_0 x, a_1 x^2, a_2 x^3, ...) \tag{2.2}$$

$$E_{flash} = h(a_0 x, a_1 x^2, a_2 x^3, ...) \tag{2.3}$$

where E_{system} is the energy consumption of the system due to filesystem related operations, E_{cpu} and E_{flash} are the energy consumption by processor and flash respectively, due to filesystem related operations. This energy consumption again, is some function of number of bytes x read or written to the flash during filesystem operations. The aim is to find what kind of relations exist in (2.1) for various filesystem related system calls. For example, the equations in (2.2, 2.3) could have a linear relation (having coefficients of higher powers of x equal zero) for creating a directory, *chdir*, proportional to number of bytes written into the filesystem while creating a directory.

C. Experimental Setup

The experimental setup consists of a StrongARM processor based system running Linux. The version of Linux is 2.4.18 with ARM processor specific patches added for the LART board. We used a LART board [20]. LART is a PC104 based Single Board Computer (SBC) running on Intel StrongARM SA1100 processor [21]. A figure of this development board along with processor, flash chips, RAM and power supply is shown in 1. LART

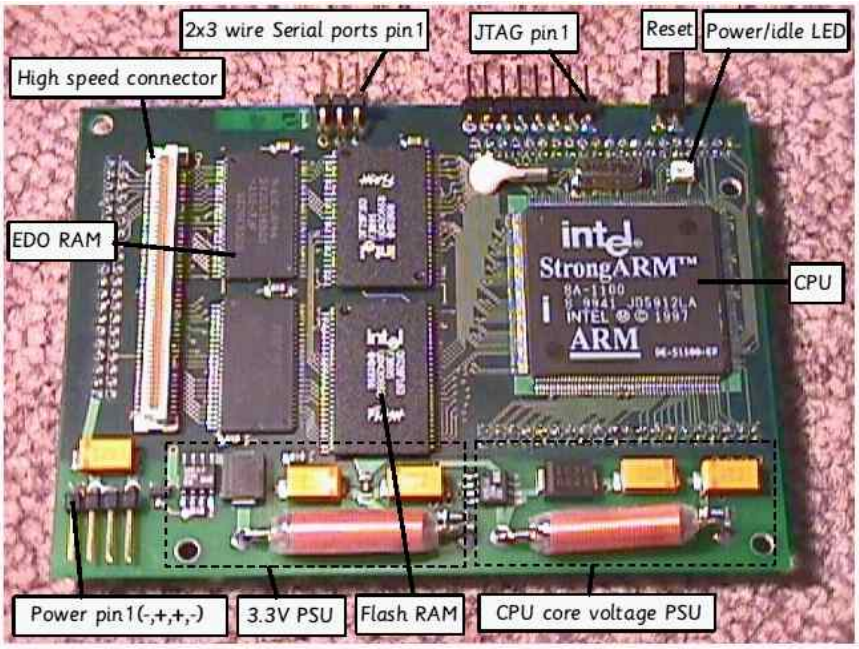

Fig. 1. LART Development Board

consumes less than 1 Watt of power and runs at around 250 MIPS. The configuration that we have has 32MB DRAM and 4MB of Flash ROM. As can be seen in 1, there are two distinct Power Supply Units (PSU) to the board. A variable voltage supply goes to the CPU and a fixed voltage supply of 3.3V drives other components of the board including the flash chips. The CPU power supply unit has the ability to generate variable voltages, however, for our experiments we used a fixed 1.5V voltage supply as our experiments did not require voltage scaling. The flash is manufactured by Intel and is part of its 3V fast boot block flash memory [1]. The JTAG port is useful to flash a bootloader if the existing bootloader on the flash gets accidently destroyed or overwritten. The image is flashed on the board by connecting the host computer's parallel port to the JTAG port using a JTAG dongle. JFlash utility can be used to download the image [20]. The Power/Idle LED glows during the period of any CPU operation. It can be set to glow or not in the kernel configuration file.

LART has a performance of around 100 MIPS. the 2x3 serial port pin is connected

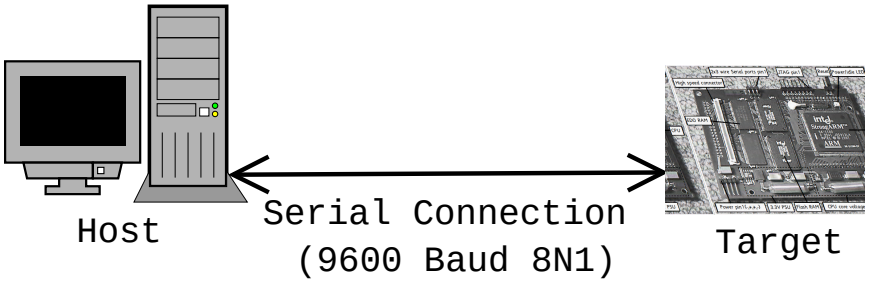

Fig. 2. Profiling Flash Accesses for Energy Measurement

to the host machine and acts as a terminal to interact with the board. A standard serial emulation program like minicom with 9600-8N1 serial settings is used.

1. Development Environment

The LART board runs on ARM (Intel SA1100) processor. Because of the limited memory and processor speed; the kernel, drivers and other application programs cannot be built on the board itself. A host based development environment exists for this reason. The host can be connected to LART board using either serial port or ethernet. The host runs on dual AMD Athlon(tm) 1.5 GHz processor. arm-linux-gcc, the cross compiler for ARM processor is used to compile the kernel, drivers and other programs on the host machine. The ARM executables are downloaded using standard serial port communication utility like Minicom. The kernel and ramdisk are also loaded on the LART board through a serial port. The host based development environment is shown in Figure 2

2. CPU Energy Measurement

LART board comes with a low value sense resistor in series with the CPU power supply to measure the power consumption of the processor. Since the value of voltage drop across this sense resistor is extremely small (in the order of millivolts), a standard differential amplifier circuit based on operational amplifier is used to amplify the output 3. The output

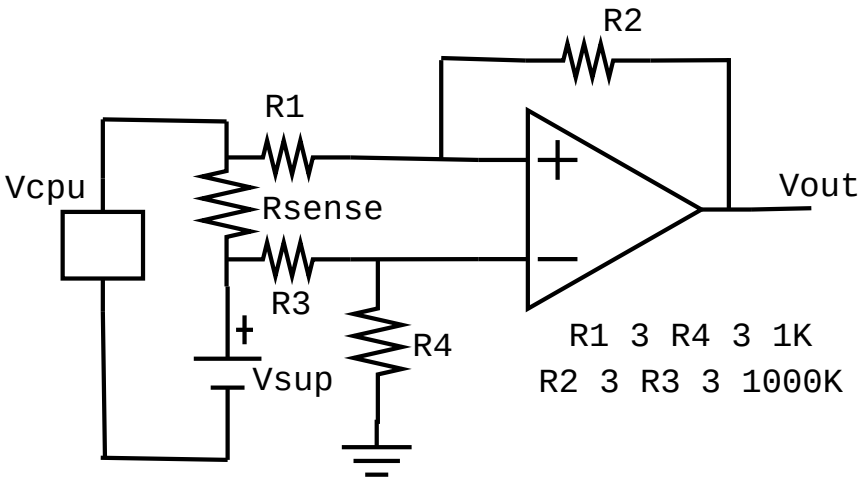

Fig. 3. CPU Power Measurement

of the differential amplifier is given by

$$V_{out} = \frac{R_2}{R_1} * (V_2 - V_1) \tag{2.4}$$

where V_2 - V_1 is the difference in voltage across the sense resistor, R_{sense}.

The actual measurement of energy consumption is done using Labview. The output of differential amplifier circuit is fed into a SCB-68 connector board which interfaces with Labview via a PCI based Data Acquisition Card (DAQ) as shown in 4. The Labview interface is configured to read data from the PCI interface at 100 scans per second. In order to start measuring energy in real-time, the setup is triggered to start by sending a pulse from the processor GPIO pins to one of the pins of the connector board. Specifically, one of the GPIO pins of the StrongARM CPU is connected to pin number 68 of the SCB-68 connector board. The start and stop signals toggle with every pulse sent through the GPIO pin. A loadable module has been written as a /proc/trigger interface, to set GPIO pins to high and low. The steps involved in profiling a section of code could be summarized as follows

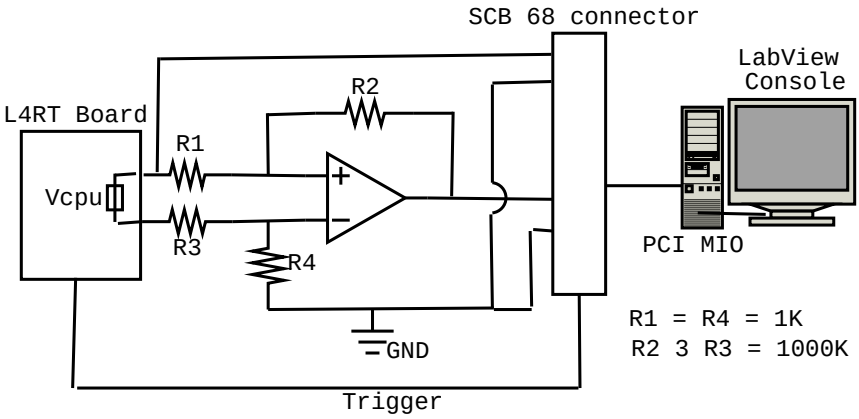

Fig. 4. LART Energy Measurement Setup

1. echo 0 > /proc/trigger
2. echo 1 > /proc/trigger
3. echo 0 > /proc/trigger
4. execute code to be profiled for energy consumption
5. echo 0 > /proc/trigger
6. echo 1 > /proc/trigger
7. echo 0 > /proc/trigger

The first three trigger signals would start the energy measurements. After the code to be profiled has run, the last three lines would send a pulse that would stop energy measurements. After this, the energy consumed in milli-Joules can be read from the Labview interface.

The instantaneous power consumption by the above circuit at any time *t* is given by

$$I(t) = \frac{V_{sense}(t)}{R_{sense}} \tag{2.5}$$

The Labview software basically integrates the power measured during the start and stop intervals (seperated by the two triggers), which is exactly the energy consumption during that interval given by

$$E_{cpu} = \int_{TRIG_{start}}^{TRIG_{stop}} I(t) * V_{dd}\ dt \tag{2.6}$$

3. Flash Energy Measurement

The energy consumption of flash is calculated using a traces taken from flash accesses. We added code inside the flash driver that calcuates the energy consumption per process depending on how much time the flash was accessed and in what mode. The energy consumption per process is calculated using the following equation

$$E_{flash} = V_{dd} * I_{mode} * t_{access} \tag{2.7}$$

The value of I_{mode} is obtained from [1] depending on what kind of operation is being done (read/write/program/erase). The time of flash access, t_{access} is calculated in microsecond resolution. This data is logged in a */proc* interface based on a per process energy consumption. The power characteristics of a Intel flash chip operating at 2.7V is shown in table below.

Table I. Power Characteristics for Flash [1]

Parameter	Current	Units
Read Current	45	mA
Program Current	8	mA
Standby Current	30	uA

The overall procedure for profiling and calculating flash energy consumption is given

in figure 5. As shown in the figure, the profiler is a layer on top of the flash driver. It profiles the write and read requests separately and outputs information on two different files in proc-filesystem. The values in procfs are logged per process along with the amount of bytes written or read. This data is used for generating a macro-model that characterizes flash energy consumption.

The 4 MB of NOR flash on the LART board has three partitions for boot-loader (128 KB), Linux kernel (896 KB) and a 3MB partition for the actual filesystem. The root partition is loaded as a ramfs partition into the RAM. The kernel uncompresses a compressed ramdisk image that has the root partition. This is done during bootup and the compressed ramdisk image is loaded using a serial link to the board. This is done due to the limited amount of flash on the board (4MB). The 3MB partition on flash is used only for our experiemnts. This also has the advantage of having /var, /tmp and other system directories in RAM, hence gives an accurate measurement of only the energy consumption we require.

D. Methodology

In this section, the overall methodology adopted in this work to come up with a macromodel is described. An example of macromodel for deleting a file is explained here. The following steps illustrate the methodology and details

1. As the first step, we need to isolate the file deletion activity out of the rest of operating system filesystem related activities. A test program is written for this purpose. The test program triggers the labview energy measurement setup, deletes twenty test files generated for the purpose of profiling and sends a trigger to stop the energy measurement. The reason twenty files are chosen is because the time required to delete one file in case of very small sized files, is too small to measure. The energy consumption is divided by twenty in later steps to calculate the average.

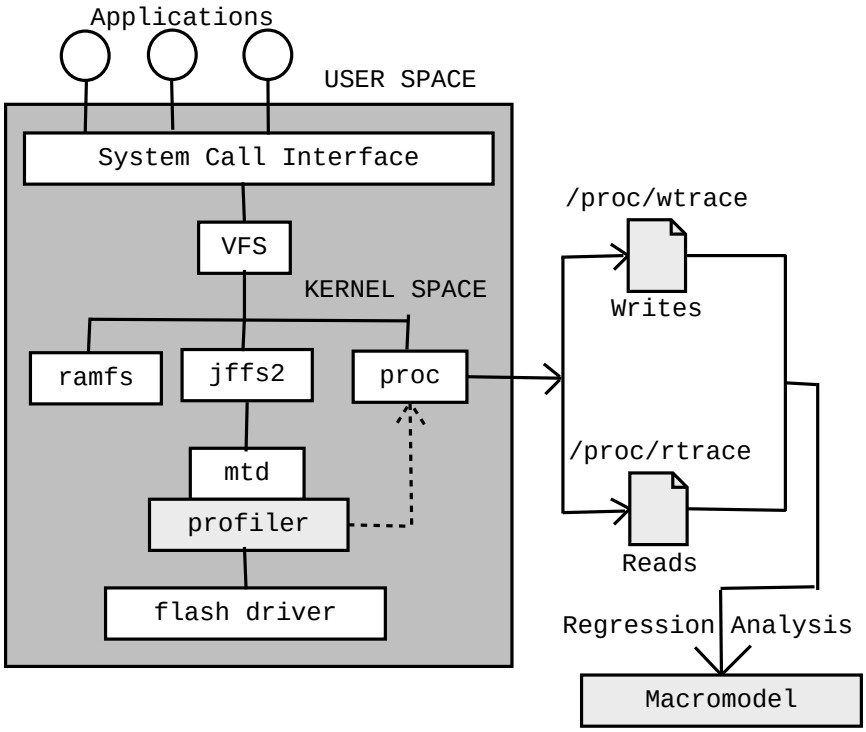

Fig. 5. Profiling Flash Accesses for Energy Measurement

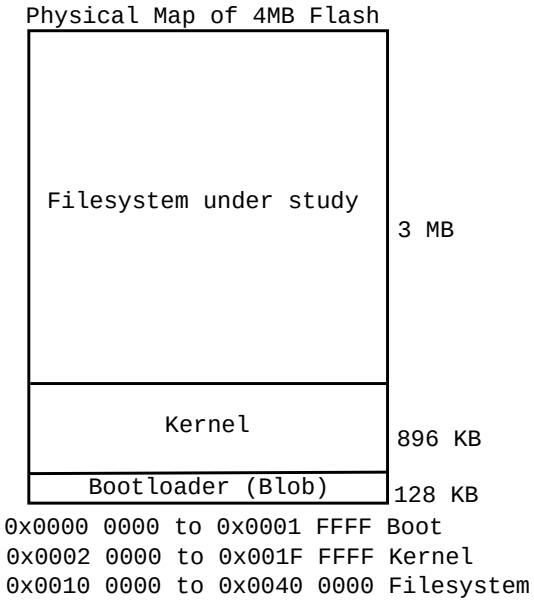

Fig. 6. Physical Map Showing Partition of Flash Address Space

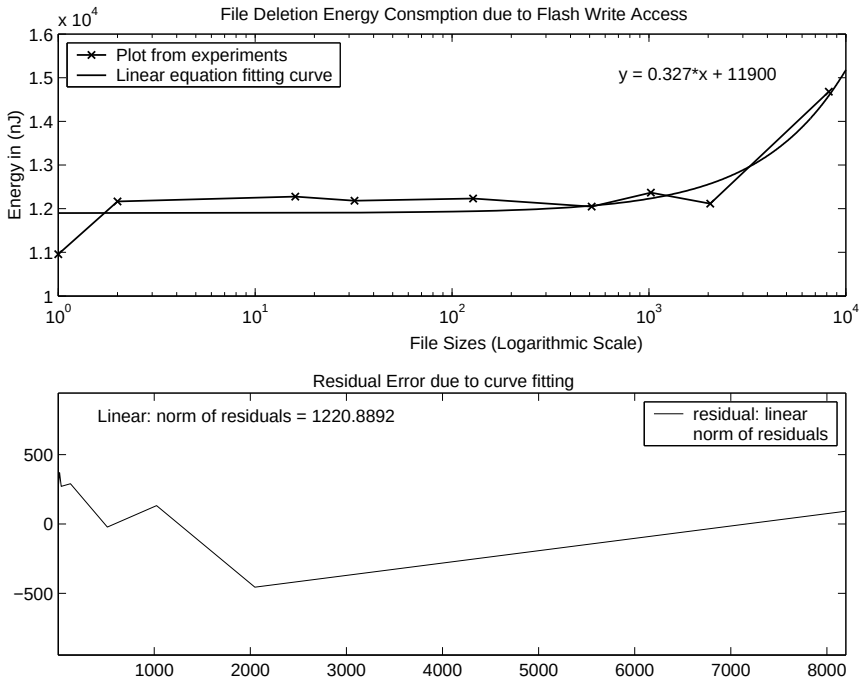

Fig. 7. Macromodeling Flash Energy Consumption Due to Write Operations

2. The average energy consumed by processer is calculated using labview. The energy consumed due to flash read and flash write accesses are calculated by taking averages out of the data in /proc/wtrace and /proc/rtrace files (corresponding to entries that belong to the process id of the test program).

3. The above two steps are repeated by varying the file size. The file size as are varied from 1 byte to 8KB in powers of two.

4. Once we have the set of energy consumption due to different file sizes, a linear equation relating energy consumption to number of bytes is calculated using equation (3.3). This equation can also be derived by plotting the values of (e_i, x_i) in matlab and using a basic fitting operation provided by matlab.

Figure 7 shows the linear equation developed for file deletion using linear regression analysis. The plot is drawn for flash write energy consumption against different file sizes on

x-axis. The lower graph shows a plot of residuals that do not fit in the equation generated out of regression analysis. This is due to the fact that the equation generated out of linear regression analysis is not a perfect fit for the observed values and is only a “best fit”. The scale on x-axis is logarithmic as file sizes are varied from 1 byte to 8192 bytes, a factor of more than 8000.

1. RAMDisk Approach

One of the problems with measuring energy consumption due to filesystem is that the operating system itself uses a number of files for locks, status and log information. These files are typically found in /var, /usr, /tmp directories. For example, the kernel messages are appended in /var/log/messages. Now, when the programs are run to measure filesystem energy consumption, the kernel should not be writing to the flash too. This would result in values of energy consumption that are more than what is actually consumed. To get around this, the root partition of the kernel is kept in RAM as a ramfs (RAM Filesystem) [22]. RAMFS is a kernel module for Linux that allows a part of RAM to look like it’s a read-write block based device.

E. Conclusion

We have formulated the problem to be addressed in this work. The high level macromodel equation and how it can be used in the context of current problem of finding energy consumption of filesystems due to processor and flash energy consumption is also introduced. The experimental setup that is being used for the experiments is described in detail. Finally, the development environment describing how kernel and ramdisk are cross compiled and loaded into the LART board is described.

CHAPTER III

MACROMODELING AND DISKLESS FILESYSTEMS

A. Introduction

This section introduces the role of filesystems in embedded systems. Eventhough, the basic idea of filesystems in case of embedded systems is same as in traditional desktop systems, the media on which it resides and access type can give rise to different options to choose from and also alternate designs. Diskless systems like flash used for secondary storage are not block based devices. These are random access devices and hence having a filesystem in these systems usually means there is a below filesystem that makes it look like a block based device. This chapter discusses how this layer could affect the energy consumption. The aim of this chapter is also to show how macromodeling can be used to study filesystems (for diskless embedded systmes) in general.

B. Filesystems

Filesystems are one of the subsystem of an operating system that help it to organize and manage data stored on secondary storage device. It is responsible for providing integrity and organizing data in a hierarchy. In addition to storaing the actual data, the filesystem also stores information about file itself (eg: date, time stamps, permissions ...). This data about data is also known as meta-data. There can be many different implementation of filesystems. Some of the common filesystems used in embedded systems are jffs2, cramfs, ext3, romfs.

Linux uses a Virtual Filesystem layer on top of the actual filesystem that delegates the user level system calls to filesystem specific system calls. The VFS layer is provided so that multiple filesystems can be supported by the kernel at the same time [23].

1. Journaling Filesystems

Conventional filesystems have a static map or order in which files are actually stored on secondary storage device. This can lead to long fsck times if the system crashes without doing a normal shutdown. This problem has been addressed in a new kind of filesystem called the journaling or logging filesystem. These filesystems keep a track of changes as opposed to the contents of file in a regular filesystem. The difference between a journaled and a log structured filesystem is that journaled filesystem keeps track of only inode changes whereas log structured filesystem keeps track of both data and meta-data changes. [24, 25]. The idea behind using journaling filesystem is that users tend to use embedded devices not like “computers” but like “appliances” and hence to switch it on and off at random. Having a journaled filesystem makes it possible to have the data integrity intact by its very nature and also serves the purpose of erasing the flash regions uniformly, a concept known as “wear-levelling”.

2. Diskless Filesystems

Embedded systems have flash as secondary storage instead of a conventional hard disk drive. Flash chips are available in two types - A NOR flash that is directly accessible and NAND flash that is addressable through a 8-bit wide bus . The same bus is used for both write and read operations using separate control signals. The advantage of NOR chips over NAND chips is that it can be cleared individually which gives more control for the driver in terms of erasing a region in flash. However, NOR chips are much more expensive than NAND chips. Flash filesystems in general can be classified into two categories [26]

1. Filesystems that are designed exclusively for flash chips. These filesystems operate directly on flash memory. JFFS2 is an example of such filesystem.

2. Conventional filesystems that run on a block based device. These filesystems can

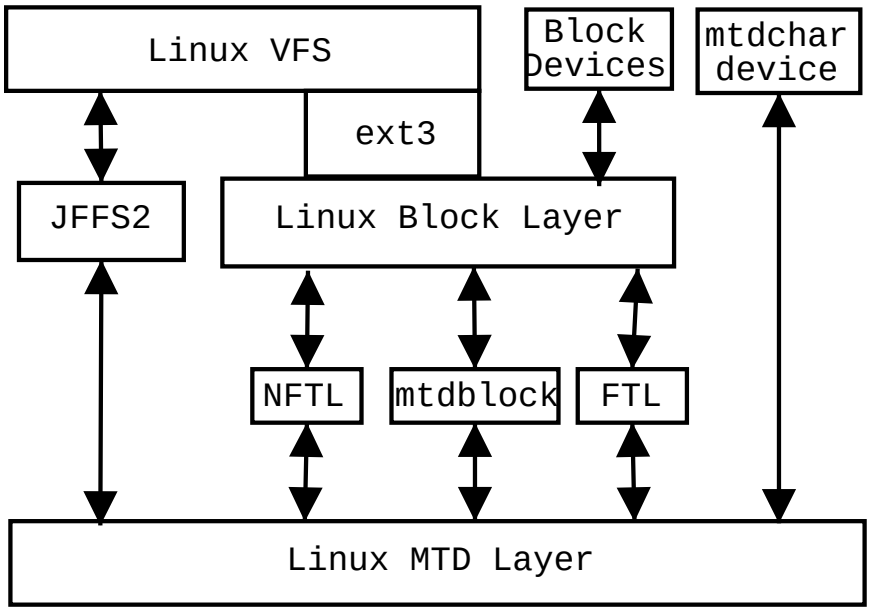

Fig. 8. Organization of a Flash Based Filesystem

use a virual block device layer as a secondary storage, which in turn operates on the flash. ext3 is an example of such a filesystem.

Figure 8 shows the organization of flash based filesystem inside Linux kernel. As can be seen, JFFS2 interacts directly withe Linux MTD layer that sits on top of the hardware specific flash driver. The Linux VFS layer on the top delegates the “generic” filesystem requests to “specific” filesystem calls. As can be seen in the figure, JFFS2 interacts directly with the flash chip through the Linux MTD layer. A conventional filesystem (eg: ext3) on the other hand, interacts with the MTD layer using intermediate layes. The layer consisting of NFTL, mtdblock, FTL are translation layer. The translation layer registers a block device with the Linux block layer, and on top of that one can mount a conventional filesystem like the ext3, cramfs, ext2 or any other filesystem for that matter. The device can be accessed by their device nodes (/dev/ftl*, /dev/nftl/*, /dev/mtdblock*) which are block special devices or using (/dev/mtd*) which is a character special device. However, the basic idea is to give the user applications on higher layer a “view” as if a block based secondary storage device

existed, eventhough the flash is actually a character based random access device.

The two different types of filesystems used in the experiments and macro-modeling are described in the next two subsections

a. JFFS2

The Journaling Flash Filesystem is a log structured filesystem initially developed by Axis Communications AB Sweeden, intended to be used on flash devices. The version two of the filesystem, popularly known as JFFS2 was developed by Redhat Inc [27]. JFFS2 like original JFFS is also a log structured filesystem. JFFS2 defines three types of nodes for the entire filesystem. The jffs2_nodetype_inode consists of all inode metadata as well as the range of data belonging to the inode. The second type of node is the jffs2_nodetype_dirent that represents a directory entry, or a link to an inode. The third and final kind of node is the jffs2_nodetype_cleanmarker which is written to a newly erased block which implies that the erase operation is finished successfully and it can be used for storage. Like any other log structured filesystem, in JFFS2 also nodes of various types are written out sequentially untile a block is fillled. At this point a new block is taken from a free_list maintained by the filesystem and writing continues. When the size of free_list reaches a threshold value, garbage collection starts. Since the amount of flash memory available in embedded systems is limited, JFFS2 compresses data while storing in the inodes. It uses zlib compression algorithm to store this compressed data. On a read operation, the compressed data in uncompressed on the fly. This prevents JFFS2 being used as an XIP (eXecute Iin Place) filesystem. However, the benefits of compressing outweigh the choice of an XIP based system.

b. EXT3

Extended Filesystem 3 (ext3) is a journaling filesystem based on the popular ext2 filesystem [28]. The journaling information in ext3 comprises of both data as well as metadata. It uses a Journaling Block Device layer, or JBD. The JBD is used to implement journaling on any kind of block device. In this way, ext3 is better than other journaling filesystems like reiserfs that journal only the metadata and not the data. However, ext3 is not designed keeping embedded systems in mind, hence the small file write performance of ext3 is not optimized. The advantages of using ext3 is that it is built on top of ext2, which has been tested in time. Also, changing the filesystem from ext2 to ext3 is fairly easy with a single command. ext3 is our second filesystem of choice for study because it follows the traditional UNIX heirarchical filesystem structure. It can be implemented in any UNIX like system that supports block device. As shown in 8, it uses lower block level facilities to interact with the secondary storage device. In our setup, we created a 3MB partition of ext3 filesystem and ran our filesystem related experiements. The mtdblocklayer as shown in 8 was used to interact with the underlying flash. This layer is responsible for calling the routines that actual do the write to the flash.

C. Macromodeling

In order to come up with a mathematical model that represents the energy consumption due to processor and flash, we ran experiments that would isolate the required filesystem related activity and made measurements of energy consumption due to processor and flash. Regression analysis was then used to formulate a linear equation relating filesystem activities to the energy consumption for applicable cases.

A mathematical model for the energy consumption analysis is developed as follows

Let $E_{cpu}(x)$ = Energy consumption of CPU due to x bytes of data in a filesystem operation.

Let $E_{fw}(x)$ = Energy consumption of flash due to x bytes of write operation to the flash.

Let $E_{fr}(x)$ = Energy consumption of flash due to x bytes of read operation from the flash.

A general equation of the above relation will be of the form

$$E(x) = f(a_0, a_1x, a_2x^2, a_3x^3, ...) \tag{3.1}$$

Since the energy consumption of the processor and the flash is directly proportional to the number of bytes that are written to or read from the filesystem, the above relation would be linear in nature. This implies that a_2 and higher terms of equation (3.1) will be 0 (i.e., $a_2 = a_3 = ... = 0$). For our purpose, we get three such independent equations representing the energy consumption due to processor, flash write operations and flash read opeartions, as shown below

$$E_{cpu}(x) = A_{cpu}x + B_{cpu} \tag{3.2a}$$

$$E_{fw}(x) = A_{fw}x + B_{fw} \tag{3.2b}$$

$$E_{fr}(x) = A_{fr}x + B_{fr} \tag{3.2c}$$

Now, the goal is to find out the unknowns in equation (3.2).

1. Macromodeling and Regression Analysis

If we have a set of n values $\{(e_0, x_0), (e_1, x_1), (e_2, x_2), ..., (e_n, x_n)\}$ that relate the energy consumption e_i to bytes x_i, using standard results from regression analysis [**?**], the relation

can be expressed in the form of a general matrix relation

$$\begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} * \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} \tag{3.3}$$

Solving the above matrix relation for unknowns A and B, we get a generalized linear equation of the form:

$$\mathbf{E}(x) = \mathbf{A}x + \mathbf{B} \tag{3.4}$$

This equation describes the energy consumption E(x) as a function of bytes x.

D. Kernel Changes

This section describes the changes made to the existing Linux kernel in order to macromodel the processor and flash energy consumption.

1. Trigger Module

The energy consumption due to processor is found out by writing programs that do certain filesystem intensive activity and finding out how much energy is consumed using the labview setup as mentioned in the previous chapter. However, the processes run for a very small interval of time (in the order of milliseconds). This requires a precise start and stop time intervals during which processor energy consumption is measured. The /proc/trigger driver is written for this purpose. It is a dynamically loadable module compiled for the same version of Linux that runs on the board. Its usage is illustrated in 9

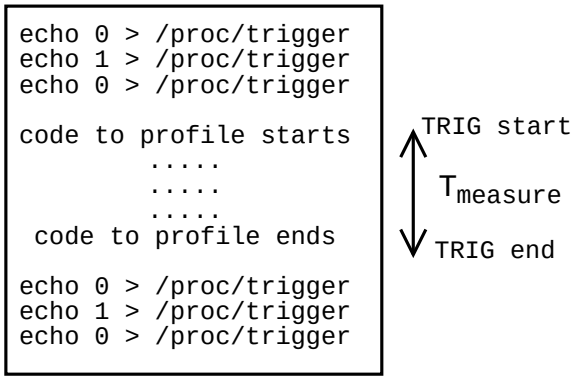

Fig. 9. Using /proc Interface to Profile a Section of Code

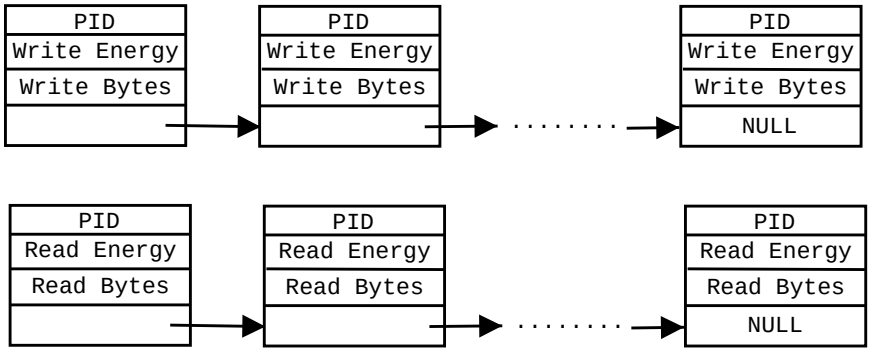

Fig. 10. Linked List Logging per Process Flash Energy Consumption

2. Flash Energy Consumption List

In order to come up with a precise energy consumption model, a per process energy consumption due to flash read and flash write operations is required. The existing kernel does not have any facility for logging flash read and write operations. Two linked lists consisting of per process flash write and flash read energy consumption are added to the kernel for this purpose. The structure of the linked list is shown in. In addition to this, read energy and write energy consumption is added to task_struct as shown in 10. The task_struct is a kernel data structure that contains all the atributes pertaining to a process. [29, 23] To access this data structure form user space, a /proc interface is used. /proc/wtrace enumerates the flash write access linked list and similarly /proc/rtrace enumerates the flash read access linked

list. A user space program can make the elements of linked list zero by doing either a ‘echo 0 > /proc/wtrace‘ or ‘echo 0 > /proc/rtrace‘.

E. Conclusions

In this section, the details of filesystem running on flash based devices were discussed. Also, the filesystem macromodeling step was explained in detail. The general mathematical equation that would be used to come up with the macromodel was derived from standard results of linear regression analysis. The kernel level changes required to implement this scheme was also discussed. This forms the stage for profiling the filesystem activities on our embedded system.

CHAPTER IV

EXPERIMENTS RESULTS

A. Introduction

This section describes the results obtained from macromodeling. We considered two filesystems for our case studies, Journaling Flash Filesystem (JFFS2) and Extensible Filesystem-3 (ext3). JFFS2 is a log structured filesystem developed for flash based devices [27]. Extensible filesystem-3 (ext3) is a journaling filesystem based on conventional ext2 filesystem. It adds journaling information to the non-journaling ext2 filesystem. [28]. The reason for using ext3 and JFFS2 was to compare two filesystems that have the same design goals of having journal information available in the filesystem along with data and metadata in order to improve availablity and robustness. Embedded devices are used as "appliance" rather than as a "computer", therefore they are prone to several abrupt power down. This makes it imperative that these systems have a journaling or a log structured filesystem to improve reliability [24]. A conventional filesystem does not update the changes made to filesystem data and metadata to the secondary storage device as soon as the changes are made. These changes are made only at a specified frequency by a kernel thread running in the background. The disadvantage with this kind of mechanism is that a shutdown made before the sync interval would leave the filesystem in an inconsistent state. This was also the reason a non-journaling filesystem was not studied.

The flash chip was partitioned to have a 3MB space for the filesystem under study.

B. Results

Tables 2 and 3 show the filesystem activities of JFFS2 and ext3 from the perspective of a user related filesystem operations. As can be seen from the tables, the energy consumption

of JFFS2 filesystem is better than that of ext3. The reason for this observation is that JFFS2 is a filesystem designed exclusively for flash devices. It works directly with flash chip driver to issue read/write requests of the required number of bytes. Ext3 on the other hand is a filesystem designed for block based device. It sees the flash as a block device and uses a translation layer called mtdblock, to issue the requests to the flash chip. Thus the read/write requests are made in multiples of block size of 128K. This implies that ext3 has a poor performance on flash for small read/write requests. This happens to be the case in a number of situations where either a small number of bytes is written to a file or the more frequent case that involve changing metadata of filesystem (eg: chmod, chown, unlink).

The energy consumption due to processor however is higher in case of JFFS2. This can be attributed to the fact that JFFS2 tries to compress data being written into flash during a write operation. While doing a read operation, it decompresses the data on fly. This is done because typically, embedded systems have a constraint on the amount of available flash, due to cost considerations. The test programs used to generate the macromodel used random data to create files, so that the compression is not optimal and the equations give a worst case bound on the energy consumption.

Tables 4 and 5 show the distribution of energy consumption for JFFS2 and ext3 due to various filesystem related system calls. The system call level energy consumption is an important metric because all application level, filesystem related functions are converted into system call by the operating system. Besides this, a breakup of system call level system call energy consumption can help in developing a low level application independent tool. The observations and conclusions derived from the previous table is also applicable for the system call level energy consumtpion of JFFS2 and ext3. It can be seen that the write energy consumption of ext3 is around 10 times greater than JFFS2. The read energy consumption of ext3 is on an average 10-90 times greater than that of JFFS2.

C. Analysis

The analysis of energy consumption due to processor is shown in Figure(**??**). It compares the processor energy consumption due to JFFS2 with worst case compression that is generated by writing random data into a file, JFFS2 with compression using data that has uniformity and ext3. The file creation activity is taken in this case, which is a most common write activity performed by a filesystem. The following results can be established from this figure

1. JFFS2 is not suited for small file size (¡ 100 bytes) due to the fact that the overhead due to compression is of the order of file size itself. For small sizes ext3 is better off by storing the data “as-is”. The advantages due to compression are significant and visible as the file sizes increase. For large files, the overhead due to compression is insignificant compared to the actual data.

2. JFFS2 without compression and ext3 consume almost the same amount of processor energy for large file sizes. JFFS2 consumes slightly higher in this case because processor cycles are wasted in trying to compress data. However, this is a worst case upper bound only for data that is random and cannot be compression and not likely to occur frequently in normal filesystem activities.

The analysis for flash write energy consumption for the same activity of creating a new file is shown in 12. The following conclusions can be drawn

1. Ext3 is expensive for small files ($<$ 128K). This is due to the fact that the requests sent to the flash chip are in multiples of 128K. For small files, this would mean that 128K bytes of data is written, no matter what is the size of request (if it happens to

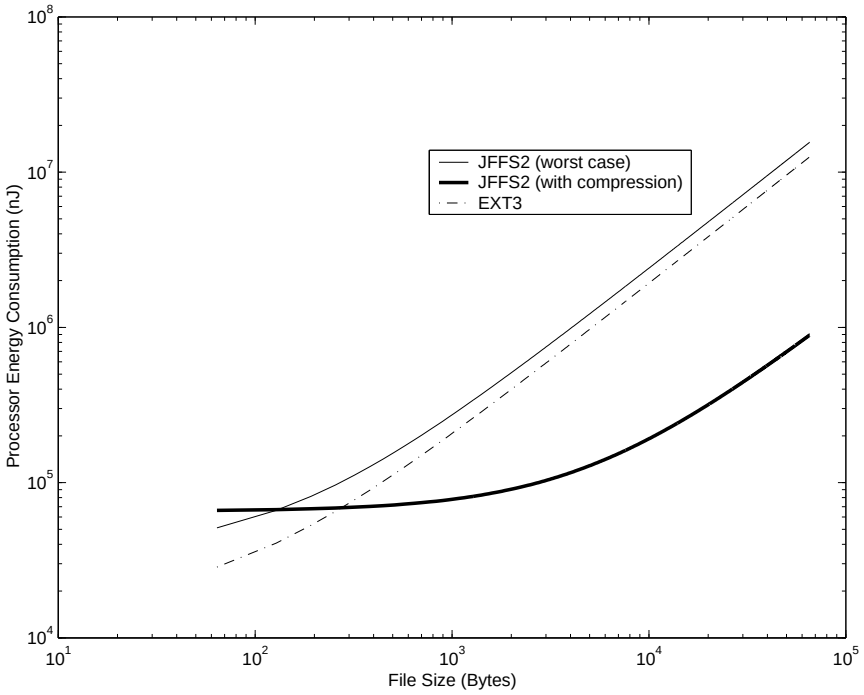

Fig. 11. Comparision of Processor Energy Consumption

be less than 128K). For large files, ext3 is almost as good as JFFS2 with best case compression.

2. JFFS2 with worst case compression is always more expensive than ext3 and JFFS2 with compression. This is due to the extra overhead of keeping the compression information along with the file itself.

Also, as a general observation, it can be seen that not all filesystem related operations are a function of number of bytes. This is due to the fact that most of the filesystem related operations involve changing the metadata. The flash is a random access device, so the seek time that typically varies in case of a hard disk drive is constant in this case.

Since we have used a regression analysis based technique to comeup with the macro-model, there are bound to be errors in the calcuated and the actual values. This error as

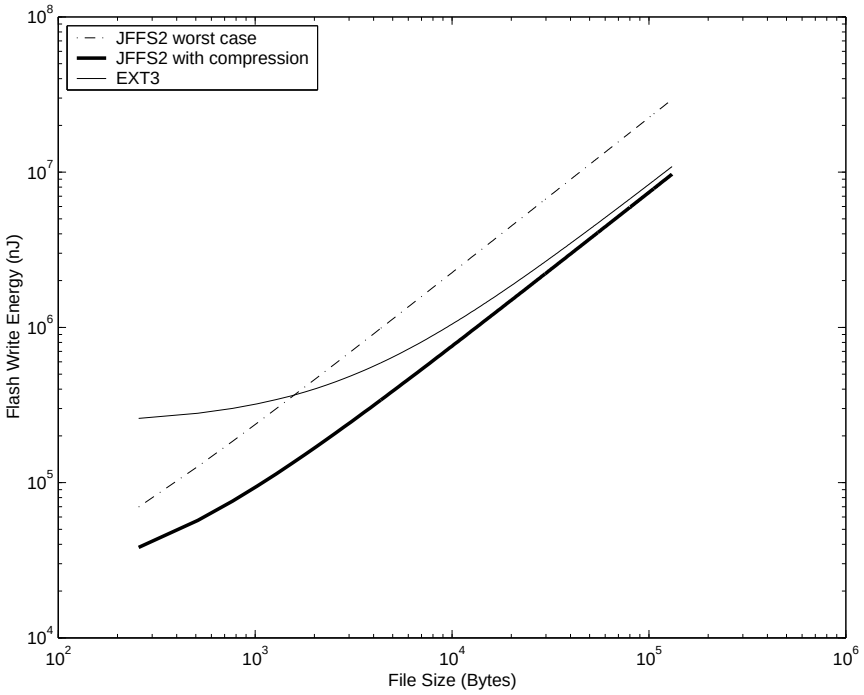

Fig. 12. Comparision of Flash Write Energy Consumption

depicted in the tables is calculated by the following formula from the results of regression analysis.

$$error = \sqrt{\sum_{1}^{n} \frac{1}{n} \left(\frac{E_m - E_a}{E_a}\right)^2} \tag{4.1}$$

E_m = The measured energy using macromodel equations

E_a = The actual energy from measurement or simulation

The error at each sampled point is calculated for all the n measurements.

D. Benchmark Programs

In order to validate the error in energy values with actual applications, three benchmark programs were run. compress uses the standard compression algorithm in order to compress a given text file and write the compressed data into an output file. v42 is a modem

based algorithm that encodes data. The output data of this program is redirected to a text file. The third benchmark program, ucbqsort, is the quicksort algorithm that reads an input file and outputs the data onto an output file. The flash write and read energy in each of these benchmark programs is calculated using both macromodel and flash trace profile. The error calcuated between the simulation and macromodel is also given in the table 6.

Table II. Energy Consumption Due to Filesystem Operations for JFFS2

Filesystem Activity	CPU Energy (nJ)	Error	Flash Energy Consumption (nJ)			
			Due to Writes	Error	Due to Read	Error
Create New File	237x + 35973	0.31	224x + 12270	1.06	0.30x + 224	0.23
Delete File	11x + 22074	0.17	0.327x + 11900	0.10	0.003x + 195	0.73
Create Directory	38000	0.82	20072	0.65	140	0.22
Remove Directory	21500	0.12	12119	0.43	38	0.64
Move a File	0.328x + 35240	0.00	0.13x + 20290	0.27	229	0.04
Copy a File	10x + 38604	0.13	176x + 37127	0.03	345	0.10
chmod	29500	0.02	13273	0.11	136	0.23

Table III. Energy Consumption Due to Filesystem Operations for EXT3

Filesystem Activity	CPU Energy (nJ)	Error	Flash Energy Consumption (nJ)			
			Due to Writes	Error	Due to Read	Error
Create New File	191x + 16321	0.30	81x + 238550	0.47	6.6x + 39044	0.37
Delete File	1.5x + 15515	0.90	6.8x + 32859	1.66	0.72x + 11007	1.13
Create Directory	292250	0.11	222789	0.16	27477	0.05
Remove Directory	17500	0.90	164279	0.24	20609	0.09
Move a File	0.68x + 17364	0.09	1.8x + 255230	0.35	0.67x + 28069	0.44
Copy a File	0.33x + 18290	1.33	507x + 208390	0.53	7.3x + 18038	0.64
chmod	16000	0.12	94485	0.04	13745	0.09

Table IV. Energy Consumption of Filesystem Related System Calls for JFFS2

System Call	CPU Energy (nJ)	Error	Flash Energy Consumption (nJ)			
			Due to Writes	Error	Due to Read	Error
creat	21750	0.09	21211	0.12	2122	0.0
link	23750	0.09	8984	0.24	169	0.31
chown	20000	0.15	13258	0.12	1083	0.0
mkdir	19750	0.02	21332	0.06	2068	0.0
rmdir	23250	0.02	12187	0.07	166	0.05
mknod	18250	0.00	21198	0.07	169	0.11
mkfifo	21750	0.15	21070	0.07	228	0.03
write	12.67x + 48885	0.02	178x + 30496	0.06	0.03x + 169	0.03
rename	33500	0.01	18870	0.21	227	0.26
unlink	93x - 27755	0.40	0.25x + 11918	0.89	90	0.10

Table V. Energy Consumption of Filesystem Related System Calls for EXT3

System Call	CPU Energy (nJ)	Error	Flash Energy Consumption (nJ)			
			Due to Writes	Error	Due to Read	Error
creat	18000	0.17	262144	0.31	13745	0.24
link	17775	0.15	169887	0.25	20608	0.21
chown	15750	0.02	95572	0.12	13746	0.22
mkdir	19750	0.06	195883	0.08	20548	0.00
rmdir	19500	0.05	247599	1.06	27481	0.59
mknod	18550	0.10	205357	0.15	27490	0.04
mkfifo	19000	0.13	165325	0.10	20605	0.07
write	80x + 14320	0.60	75x + 242850	0.56	20645	0.01
rename	18500	0.06	247832	1.02	27487	0.50
unlink	1.8x + 14320	0.29	5.2x + 24210	0.66	9550	0.16

Table VI. Energy Consumption of Filesystem for Programs in JFFS2

Benchmark	Flash Write Energy(nJ)			Flash Read Energy (nJ)		
	Traced	Evaluated	Error	Traced	Evaluated	Error
compress	62011	59742	-3.6	340	358	5.2
ucbqsort	139693	147376	5.2	335	358	6.8
v42	372899	385900	3.4	153	162	-7.0

CHAPTER V

FUTURE WORK

A. Exploring Dynamic State of Flash

Most flash chips provide a “standby state” that consumes current in the order of uA. This is order of around 1000 less than that of normal operation where the current is in mA. Depending on the access patterns, it can be calcuated at what times the flash has low energy consumption and accordingly, the flash can be put in the low power, standy state. This is similar to ACPI hard disk drives in conventional desktop systems that can switch to a low-power “sleep state” when not in use. One of the ways to implement this can be using the scheduler, that can me modified and made energy aware so that it switches state of the flash chip.

B. Study of Other Filesystems

The work currently studies only two of the filesystems, JFFS2 and ext3. One has higher processor energy consumption and the other has higher flash energy consumption. The tool can be used to design filesystem that does not use the mtdblock (as in the case of JFFS2), but uses the other features of ext3.

C. Efficient Flash Partitioning Scheme

The energy consumption can be studied to partition flash such that all the libraries and other read-only directory structure is kept on a partition that has low flash read-energy consumption. The rest of the filesystem can be made a read-write partition depending on energy requirements. This partition can be log structured to improve the reliability.

CHAPTER VI

CONCLUSION

In this thesis, a mathematical model describing energy consumption of filesystem is characterized. The energy consumption due to processor, flash write operations and flash read operations are quantified using linear regression analysis. This study is done for two different types of filesystems that are popularly used. The results of the two filesystems are compared to find out the advantages and disadvantages of each.

The mathematical modeling describing energy consumption as a function of filesystem level system-calls can be a powerful tool for system designer to make decisions about filesystem of choice from an energy consumption point of view. It can also be used to find out the energy consumption due to filesystem overhead in user applications.

REFERENCES

[1] Intel Flash Memories, "Intel Fast Boot Block Flash Memory," in http://www.intel.com/design/ flash/support/DevChar/28f800f3.htm, May 2003.

[2] John Zedlewski Sumeet, "Modeling hard-disk power consumption," in *Proc. Second Conference on File and Storage Technologies*, March, 2003.

[3] Jonathan T. Moore, Michael Hicks, and Scott Nettles, "General-purpose persistence using flash memory," Technical report ms-cis-97-3, Department of Computer and Information Science, University of Pennsylvania, 1997.

[4] J. Eager, Ed., *Advances in rechargeable batteries spark product innovation*, Silicon Valley Computer Conference, Santa Clara, CA, August 1992.

[5] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power cmos digital design," in *Low-Power CMOS Digital Design. JSSC*, pp. 473–484, April, 1992.

[6] Neil H. E. Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*, 2nd ed. Upper Saddle River, NJ: Pearson Education, 1992.

[7] Amit Sinha, "Energy aware software," M.S. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.

[8] A. Vahdat, A. Lebeck, and C. Ellis, "Every joule is precious: The case for revisiting operating system design for energy efficiency," in *SIGOPS European Workshop*, Kolding, Denmark, 2000, pp. 31-36.

[9] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 437–445, 1994.

[10] Robert P. Dick, Ganesh Lakshminarayana, Anand Raghunathan, and Niraj K. Jha, "Power analysis of embedded operating systems," in *Design Automation Conference*, San Diego, CA, 2000, pp. 312–315.

[11] Jason Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *Symposium on Operating Systems Principles*, 1999, pp. 48–63.

[12] F. Bellosa, "OS-directed throttling of processor activity for dynamic power management," Technical report tri4 -3-9, Department of Computer Science, University of Erlangen, Germany, 1999.

[13] T. Šimunić, L. Benini, and G. D. Micheli, "Event-driven power management of portable systems," in *International Symposium on System Synthesis*, 1999, pp. 18–23.

[14] N. Jha T. K. Tan, A. Raghunathan, "Embedded operating system energy analysis and macro-modeling," in *ICCD*, 2002.

[15] T. K. Tan, Anand Raghunathan, Ganesh Lakshminarayana, and Niraj K. Jha, "High-level software energy macro-modeling," in *Design Automation Conference*, 2001, pp. 605–610.

[16] Amit Sinha and Anantha Chandrakasan, "Jouletrack - a web based tool for software energy profiling," in *Design Automation Conference*, 2001, pp. 220–225.

[17] N. Jha T. K. Tan, "EMSIM an energy simulation framework for an embedded operating system," in *Proceedings of International Symposium on Circuit and Systems*, May 2002.

[18] David Brooks, Vivek Tiwari, and Margaret Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *ISCA*, 2000, pp. 83–94.

[19] Doug Burger, Todd M. Austin, and Steve Bennett, "Evaluating future microprocessors: The simplescalar tool set," Tech. Rep. CS-TR-1996-1308, 1996.

[20] LART, Delft University of Technology, "The lart pages," in http://www.lart.tudelft.nl/, May, 2003.

[21] Intel PCA Processors "Intel strongarm processor," in http://www.intel.com/design/pca/applicationsprocessors/1110_brf.htm, May 2003.

[22] Handhelds.org, The handheld portal, "RAMFS," in http://www.handhelds.org/z/wiki/RAMfs, May 2003.

[23] Cesati M Bovet P. D., *Understanding the Linux Kernel*, 1st ed. Sebastopol, CA: O'Reilly, 1998.

[24] Mendel Rosenblum and John K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[25] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin, "An implementation of a log-structured file system for UNIX," in *USENIX Winter*, 1993, pp. 307–326.

[26] Aleph One Limited, "JFFS2 and NAND," in http://www.aleph1.co.uk/armlinux/projects/yaffs/jffs2_and_nand.htm, May, 2003.

[27] *JFFS: The Journaling Flash File System*. Ottawa Linux Symposium, 2001, in http://sources.redhat.com/jffs2/jffs2.pdf.

[28] Stephen Tweedie, "Journaling the Linux ext2fs filesystem," in *LinuxExpo '98*, 1998.

[29] Rubini, *Writing Device Drivers in Linux*, 2nd ed. Sebastopol, CA: O'Reilly, 1999.

APPENDIX A

SOURCE CODE FOR TRIGGER DRIVER

```
/*
 * trigger.c: A module that sends a high or low to one of GPIO pins of LART
 *
 * Author:
 * - Siddharth Choudhuri (choudhri@cs.tamu.edu): final version
 *
 * Goal of the program
 * ===================
 * The goal of this driver is to set pin 16 (GPIO 3) to high or low to
 * start the trigger for NI Labview for power measurements
 *
 * Usage
 * =====
 * echo 0 > /proc/trigger  -> To set GPIO Pin 3 to LOW
 * echo 1 > /proc/trigger  -> To set GPIO Pin 3 to HIGH
 *
 */

#include <linux/config.h>
#include <linux/kernel.h>/* because this is kernel level program */
#include <linux/module.h>/* because it is a module */
#include <linux/init.h>/* for the __init macros */
#include <linux/proc_fs.h>/* for the proc filesystem entry */
#include <linux/ioport.h>
#include <asm/uaccess.h>/* to copy to/from userspace */
#include <asm/arch/hardware.h>  /* to set the GPIO pins to High and Low */

#define MODULE_NAME "trigger"
#define MODULE_VERSION "1.0"

#define LCD_PPC_BITS    0x000000ff

static int gpio_pin_status;

static int read_trigger(char *, char **, off_t, int, int *, void *);
static int write_trigger(struct file *, const char *, unsigned long, void *);

static int __init init_trigger_mod(void)
{
struct proc_dir_entry *entry;

/* create a new proc entry by name trigger */
entry = create_proc_entry(MODULE_NAME, 0666, NULL);
```

```
/*entry->uid = 0;*/

/* voltage control init */
         LCCR0 &= ~LCCR0_LEN;     /* disable the built-in lcd controller */

/* Write to a read-only register ??? */
PPFR  |=  PPFR_LCD;      /* let the PPC control the lcd pins */

/* set register vaules */
PPSR  |= 0x00000000; /* all pins low */
/* set direction */
PPDR  |=  LCD_PPC_BITS; /* pins are outputs */

if (entry) {
/* callback functions to read and write to /proc/trigger */
entry->read_proc = read_trigger;
entry->write_proc = write_trigger;
}
else {
printk(KERN_ERR MODULE_NAME ": can't create /proc" MODULE_NAME "\n");
return -1;
}

return 0;
}

static int read_trigger(char *page, char **start, off_t off, int count, int *eof, void *data)
{
int len;

len = sprintf(page, "%d\n", gpio_pin_status);
/*printk("Module trigger: read %d\n", gpio_pin_status);*/

return len;
}

static int write_trigger(struct file *f, const char *buffer, unsigned long count, void *data)
{
int gpio_to_set, len;
char *p;

MOD_INC_USE_COUNT;
len = count;
gpio_to_set = simple_strtoul(buffer, &p, 0);
/*printk("Module trigger: wrote %d\n",gpio_to_set); */
gpio_pin_status = gpio_to_set;

if ( gpio_pin_status == 1 ) {
/* Set LDD7 to high */
PPSR  |= 0x00000080; /* set LDD7 to high */
printk("Module trigger: setting LDD7 to 1\n");
```

```
}
else {
/* Set LDD 7 to low */
GPCR = GPIO_GPIO3;
PPSR  &= 0xFFFFFF7F; /* set LDD7 to low */
printk("Module trigger: setting LDD7 to 0\n");
}

MOD_DEC_USE_COUNT;

return len;
}

static void __exit exit_trigger_mod(void)
{
remove_proc_entry(MODULE_NAME, NULL);

return;
}

module_init(init_trigger_mod);

module_exit(exit_trigger_mod);
```

APPENDIX B

CREATING A RAMDISK IMAGE

```
dd if=/dev/zero of=/dev/ram1 bs=1k count=4096
mkfs.ext3 -vm0 /dev/ram1 4096
mount -t ext3 /dev/ram1 /mnt/ramdisk
cp -av <prepared_filesystem> /mnt/ramdisk
umount /mnt/ramdisk
dd if=/dev/ram1 bs=1k count=4096 | gzip -v9 > ramdisk.gz
uuencode ramdisk.gz ramdisk.gz > ramdisk.gz.uu
```

APPENDIX C

CREATING A JFFS2 FLASH BASED IMAGE

```
1. Create a JFFS2 filesystem image out of a root filesystem that is located
in directory rootfs
mkfs.jffs2 -r rootfs/ -o jffs2_image.img
jffs2_image is the name of the output file that has the JFFS2 image.

2. Erase the flash partition
eraseall /dev/mtd2 -or- erase /dev/mtd2 0 0x300000
This command erases the 3MB partition on flash that is used for filesystem

3. Download the JFFS2 image (jffs2_image.img) to the LART board using z-modem
serial transfer

4. Copy the filesysetm image onto the flash partition
cat jffs_image.img > /dev/mtd2

5. Load the JFFS2 module into the kernel
modprobe -a jffs2

6. Mount the JFFS2 filesystem
mount -t jffs2 /dev/mtdblock2 /mnt
```

APPENDIX D

KERNEL DATA STRUCTURES FOR LOGGING FLASH ENERGY CONSUMPTION

```
/*
 * Trace.h
 * This file has the data structures required to generate a trace
 * based simulation of flash chip energy consumption
 *
 * Author: Siddharth Choudhuri <choudhri@cs.tamu.edu>
 */

#ifndef __MTD_TRACE_H__
#define __MTD_TRACE_H__

#include <linux/types.h>
#include <linux/list.h>

/*
 * These defines are from Intels' flash chip manual under section 8.0
 *
 * The following defines are to convert the above values to
 * integer values. Floating point seems to have problems
 * with kernel compilation as ARM has a floating point emulation only
 */
#define FLASH_CHIP_VDD 27 /* 10 x V  */
#define FLASH_READ_CURRENT 45 /* mA */
#define FLASH_PROGRAM_CURRENT 8 /* mA */
#define FLASH_ERASE_CURRENT 8 /* mA */
#define FLASH_STANDBY_CURRENT 3 /* uA */
#define UPDATE_READ 0
#define UPDATE_WRITE 1

/*
 * This is the basic structure that holds per process energy consumption
 * due to write operations to flash chip
 */
struct trace_wr_energy {
pid_t pid;              /* PID of the process*/
char comm[16];          /* Name of the process */
u64 write_energy;       /* Energy consumed (mJ) due to write to flash */
unsigned int bytes_written; /* Total number of bytes written */
};

/*
 * This is the basic structure that holds per process energy consumption
 * due to write operations to flash chip
```

```
 */
struct trace_rd_energy {
pid_t pid;        /* PID of the process*/
char comm[32];    /* Name of the process */
u64 read_energy;  /* Energy consumed (mJ) due to read from flash */
unsigned int bytes_read;    /* Total number of bytes read */
};

/*
 * This to generate a linked list out of the trace_energy
 * using the list_head provided by the kernel in <linux/list.h>
 * This is for write operations
 */
struct trace_wr_list {
struct trace_wr_energy tr_energy;
struct trace_wr_list *next;
struct trace_wr_list *prev;
};

/*
 * This to generate a linked list out of the trace_energy
 * using the list_head provided by the kernel in <linux/list.h>
 * This is for read operations
 */
struct trace_rd_list {
struct trace_rd_energy tr_energy;
struct trace_rd_list *next;
struct trace_rd_list *prev;
};


#endif /* __MTD_TRACE_H__ */
```

APPENDIX E

READINGS FOR ENERGY MEASUREMENT

This table illustrates the readings for processor and flash energy for different file sizes. The file delete activity using rm command was being profiled. Similar tables were created for each activity whose macromodel was made.

Table VII. Energy Consumption Characterics Due to File Delete

File Size	Bytes Written	Write Energy	Bytes Read	Read Energy	CPU Energy
1	61	10956	36	179.7	0.0275
2	68	12166	40	204.9	0.0270
8	40	12280	38.8	116.35	0.024
16	68	12275	40	224.5	0.0265
32	68	12182	40	211.2	0.0280
128	68	12232	40	208.7	0.0285
512	68	12046	40	235.5	0.0270
1024	68.6	12368	40.6	197.75	0.0225
2048	68	12116	40	190	0.0225
8196	81.2	14679	53.2	224	0.1195

VITA

Siddharth Choudhuri received his undergraduation degree in Computer Science and Engineering from University College of Engineering, Sambalpur University in 1999. He worked as a Senior Design Engineer at GE Medical Systems, writing drivers for Magnetic Resonance Imaging (MRI) scanners before beginning study for the MS in Computer Engineering at Texas A&M University in 2001.

Permanent Address
Dr. P. C. Choudhuri
215 Dharmavihar,
Bhubaneswar 751030
INDIA

The typist for this thesis was Siddharth Choudhuri.